# 14

# Essential Shell Programming

The activities of the shell are not restricted to command interpretation alone. The shell has a whole set of internal commands that can be strung together as a language. We need two chapters to discuss this language. In this chapter, we focus on the Bourne shell—the lowest common denominator of all shells. We have reserved the discussion of the advanced features of the Korn and Bash shells for Part II of the book. However, everything discussed here applies to both these shells too. The C shell uses totally different programming constructs and has been separately treated in Appendix A.

A shell program runs in *interpretive* mode. It is not compiled into a separate executable file as a C program is. Each statement is loaded into memory when it is to be executed. Shell scripts consequently run slower than those written in high-level languages. However, what makes shell programs powerful is that external UNIX commands blend easily with the shell's internal constructs. Speed is not a factor in many jobs we do, and in many cases, using the shell is an advantage—especially in system administrative tasks. *The UNIX system administrator must be an accomplished shell programmer.*

## WHAT YOU WILL LEARN

- How shell scripts are executed, and the role of the *interpreter line*.
- Make shell scripts interactive using **read**.
- Use *positional parameters* to read command line arguments.
- The role of the **exit** statement in terminating a script.
- Elementary decision making with the || and && operators.
- Comprehensive decision making with the **if** conditional.
- Use **if** in tandem with **test** to perform numeric and string comparison, and test a file's attributes.
- Use the wild-card pattern matching features of **case** for decision making.
- Integer computing and string handling using **expr**.
- Use a **while** loop to repeatedly execute a set of commands.
- Use a **for** loop with a list.
- Manipulate the positional parameters with the **set** and **shift** statements.
- Use **trap** to control the behavior of a shell script when it receives signals.

## TOPICS OF SPECIAL INTEREST

- Redirect some statements to /dev/tty so the other statements can be manipulated collectively
- Exploit the hard linking feature with $0 to make a script behave in different ways depending on the name by which it is invoked.
- Use a *here document* as the fourth source of standard input to run any interactive shell script noninteractively.

## 14.1 SHELL SCRIPTS

When a group of commands have to be executed regularly, they should be stored in a file, and the file itself executed as a **shell script** or **shell program**. Though it's not mandatory, we normally use the .sh extension for shell scripts. This makes it easy to match them with wild-cards.

Shell scripts are executed in a separate child shell process, and this sub-shell need not be of the same type as your login shell. In other words, even if your login shell is Bourne, you can use a Korn sub-shell to run your script. By default, the child and parent shells belong to the same type, but you can provide a special *interpreter line* in the first line of the script to specify a different shell for your script.

Tip: Generally, Bourne shell scripts run without problem in the Korn and Bash shells. There are two issues in Bash, however. First, Bash evaluates $0 differently. This has to be handled by appropriate code in the script. Second, it doesn't recognize escape sequences used by **echo** (like \c and \n) unless the -e option is used. To make **echo** behave in the normal manner, place the statement **shopt -s xpg_echo** in your rc file (probably, ~/.bashrc).

Use your **vi** editor to create the shell script, **script.sh** (Fig. 14.1). The script runs three **echo** commands and shows the use of variable evaluation and command substitution. It also prints the calendar of the current month.

```
#!/bin/sh
# script.sh: Sample shell script
echo "Today's date: `date`"
echo "This month's calendar:"
cal `date "+%m 20%y"`              # This month's calendar
echo "My shell: $SHELL"
```

<p align="center">Fig. 14.1  script.sh</p>

Note the comment character (#) that can be placed anywhere in a line; the shell ignores all characters placed on its right. However, this doesn't apply to the first line which also begins with a #. This is the **interpreter line** that was mentioned previously. It always begins with #! and is followed by the pathname of the shell to be used for running the script. Here, this line specifies the Bourne shell. To run the script, make it executable first and then invoke the script name:

```
$ chmod +x script.sh
$ script.sh
Today's date: Mon Jan  6 10:02:42 IST 2003
This month's calendar:
     January 2003
Su Mo Tu We Th Fr Sa
          1  2  3  4
 5  6  7  8  9 10 11
12 13 14 15 16 17 18
19 20 21 22 23 24 25
26 27 28 29 30 31
```

*PATH must include the
or else use ./script.sh*

```
My shell: /bin/sh
```

The script is too simple to need any explanation—no inputs, no command line arguments and control structures. We'll be progressively adding these features to our future scripts.

We mentioned that shell scripts are executed by a child shell. Actually, the child shell reads and executes each statement in sequence (in interpretive mode). You can also *explicitly* spawn a child of your choice with the script name as argument:

```
sh script.sh
```
*Will spawn a Bourne shell*

When used in this way, the Bourne sub-shell opens the file but ignores the interpreter line. The script doesn't need to have execute permission either. We'll make it a practice to use the interpreter line in all our scripts.

---

**Tip:** The pathname of the shell specified in the interpreter line may not match the actual pathname of your system. This sometimes happens with downloaded scripts. To prevent these scripts from breaking, make a symbolic link between the two locations. Note that root access is required to make a symbolic link between **/bin/ksh** and **/usr/bin/ksh**.

---

## 14.2 read: MAKING SCRIPTS INTERACTIVE

The **read** statement is the shell's internal tool for taking input from the user, i.e., making scripts interactive. It is used with one or more variables. Input supplied through the standard input is read into these variables. When you use a statement like

```
read name
```

the script pauses at that point to take input from the keyboard. Whatever you enter is stored in the variable name. Since this is a form of assignment, no $ is used before name. The script, empl.sh (Fig. 14.2), uses **read** to take a search string and filename from the terminal.

Shell scripts accept comments prefixed by # anywhere in a line. You know what the sequence does (5.3). Run the script and specify the input when the script pauses twice:

```
#!/bin/sh
# empl.sh: Interactive version - uses read to take two inputs
#
echo "Enter the pattern to be searched: \c"        # No newline
read pname
echo "Enter the file to be used: \c"               # Use echo -e or
read flname                                        # shopt -s xpg_echo in Bash
echo "Searching for $pname from file $flname"
grep "$pname" $flname
echo "Selected records shown above"
```

Fig. 14.2 emp1.sh

```
$ emp1.sh
Enter the pattern to be searched: director
Enter the file to be used: emp.1st
Searching for director from file emp.1st
9876|jai sharma        |director |production|12/03/50|7000
2365|barun sengupta    |director |personnel |11/05/47|7800
Selected records shown above
```

The script first asks for a pattern to be entered. Input the string director, which is assigned to the variable pname. Next, the script asks for the filename; enter the string emp.1st, which is assigned to the variable flname. **grep** then runs with these two variables as arguments.

A single **read** statement can be used with one or more variables to let you enter multiple arguments:

```
read pname flname
```

If the number of arguments supplied is less than the number of variables accepting them, any leftover variables will simply remain unassigned. However, when the number of arguments exceeds the number of variables, the remaining words are assigned to the *last* variable.

## 14.3 USING COMMAND LINE ARGUMENTS

Like UNIX commands (which are written in C), shell scripts also accept arguments from the command line. They can, therefore, run noninteractively and be used with redirection and pipelines. When arguments are specified with a shell script, they are assigned to certain special "variables"—rather *positional parameters*.

The first argument is read by the shell into the parameter $1, the second argument into $2, and so on. You can't technically call them shell variables because all variable names start with a letter. In addition to these positional parameters, there are a few other special parameters used by the shell. Their significance is noted below:

$* — It stores the complete set of positional parameters as a single string.

$# — It is set to the number of arguments specified. This lets you design scripts that check whether the right number of arguments have been entered.

```
#!/bin/sh
# emp2.sh: Non-interactive version  - uses command line arguments
#
echo "Program: $0                    # $0 contains the program name
The number of arguments specified is $#
The arguments are $*"                # All arguments stored in $*
grep "$1" $2
echo "\nJob Over"
```

Fig. 14.3 emp2.sh

$0— Holds the command name itself. You can link a shell script to be invoked by more than one name. The script logic can check $0 to behave differently depending on the name by which it is invoked.

The next script, **emp2.sh** (Fig. 14.3), runs **grep** with two positional parameters that are set by the script arguments, director and emp.1st:

```
$ emp2.sh director emp.1st
Program: emp2.sh
The number of arguments specified is 2
The arguments are director emp.1st
1006|chanchal singhvi |director |sales     |03/09/38|6700
6521|lalit chowdury   |director |marketing |26/09/45|8200

Job Over
```

When arguments are specified in this way, the first word (the command itself) is assigned to $0, the second word (the first argument) to $1, and the third word (the second argument) to $2. You can use more positional parameters in this way up to $9 (and using the **shift** statement, you can go beyond).

When you use a multiword string to represent a single command line argument, you must quote it. To look for chanchal singhvi, use **emp2.sh "chanchal singhvi" emp.1st**. You have noted this quoting requirement when using **grep** also (13.1).

All assignments to positional and special parameters are made by the shell. You can't really tamper with their values, except in an indirect fashion, but you can use them to great advantage in several ways. They will be used over and over again in shell scripts, and are listed in Table 14.1.

**Bash Shell:** $0 in Bash prepends the ./ prefix to the script name. In the example above, it would have shown ./emp2.sh instead of emp2.sh. You need to keep this in mind when you make use of $0 to develop portable scripts.

**Table 14.1** Special Parameters Used by the Shell

| Shell Parameter | Significance |
|---|---|
| $1, $2... | Positional parameters representing command line arguments |
| $# | Number of arguments specified in command line |
| $0 | Name of executed command |
| $* | Complete set of positional parameters as a single string |
| "$@" | Each quoted string treated as a separate argument (recommended over $*) |
| $? | Exit status of last command |
| $$ | PID of the current shell (9.1.1) |
| $! | PID of the last background job (9.9.1) |

# 14.4 exit AND EXIT STATUS OF COMMAND

C programs and shell scripts have a lot in common, and one of them is that they both use the same command (or function in C) to terminate a program. It has the name **exit** in the shell and **exit()** in C. We'll take up the **exit** function in Part II of this book, but in this section, we'll examine the shell's **exit** command. The command is generally run with a numeric argument:

```
exit 0        Used when everything went fine
exit 1        Used when something went wrong
```

These are two very common **exit** values. You don't need to place this statement at the end of every shell script because the shell understands when script execution is complete. Rather, it's quite often used with a command when it fails.

Once **grep** couldn't locate a pattern *(13.1)*; we said then that the command *failed*. What we meant was that the **exit** function in the **grep** code was invoked with a nonzero argument (**exit(1)**). This value (1) is communicated to the calling program, usually the shell.

It's through the **exit** command or function that every command returns an **exit status** to the caller. Further, a command is said to return a *true* exit status if it executes successfully, and *false* if it fails. The **cat** command as used below:

```
$ cat foo
cat: can't open foo
```

returns a nonzero exit status because it couldn't open the file. The shell offers a variable ($?) and a command (**test**) that evaluates a command's exit status.

**The Parameter $?**   The parameter $? stores the exit status of the last command. It has the value 0 if the command succeeds and a nonzero value if it fails. This parameter is set by **exit**'s argument. If no exit status is specified, then $? is set to zero (true). Try using **grep** in these ways and you'll see it returning three different exit values:

```
$ grep director emp.1st >/dev/null; echo $?
0
$ grep manager emp.1st >/dev/null; echo $?
1
```

Success

Failure—in finding pattern

```
$ grep manager emp3.1st >/dev/null; echo $?
grep: can't open emp3.1st                          Failure  in opening file
2
```

The exit status is extremely important for programmers. They use it to devise program logic and branches into different paths depending on the success or failure of a command. For example, there's no point in continuing with script execution if an important file doesn't exist or can't be read.

**Tip:** To find out whether a command executed successfully or not, simply use **echo $?** after the command. 0 indicates success, other values point to failure.

**Note:** Success or failure isn't as intuitive as it may seem. The designer of **grep** interpreted grep's inability to locate a pattern as failure. The designer of **sed** thought otherwise. The command **sed -n '/manager/p'** emp.1st returns a true value even if manager is not found!

## 14.5 THE LOGICAL OPERATORS && AND ||—CONDITIONAL EXECUTION

The script **emp1.sh** has no logic to prevent display of the message, Selected lines shown above, when the pattern search fails. That is because we didn't use **grep**'s exit status to control the flow of the program. The shell provides two operators that allow conditional execution—the && and ||, which typically have this syntax:

```
cmd1 && cmd2
cmd1 || cmd2
```

The && delimits two commands; the command *cmd2* is executed only when *cmd1* succeeds. You can use it with **grep** in this way:

```
$ grep 'director' emp.1st && echo "pattern found in file"
1006|chanchal singhvi |director |sales     |03/09/38|6700
6521|lalit chowdury   |director |marketing |26/09/45|8200
pattern found in file
```

The || operator plays an inverse role; the second command is executed only when the first fails. If you "grep" a pattern from a file without success, you can notify the failure:

```
$ grep 'manager' emp.1st || echo "Pattern not found"
Pattern not found
```

The || goes pretty well with the **exit** command. You often need to terminate a script when a command fails. The script **emp2.sh** can be modified to include this feature. The following two lines ensure that the program is aborted when the **grep** command fails and a message is printed if it succeeds:

```
grep "$1" $2  || exit 2
echo "Pattern found - Job Over"
```
*No point continuing if search fails.*
*Executed only if grep succeeds*

This segment makes rudimentary decisions which the previous scripts couldn't. In fact, the && and || operators are recommended for making simple decisions. When complex decision making is involved, they have to make way for the **if** statement.

## 14.6 THE if CONDITIONAL

The if statement makes two-way decisions depending on the fulfillment of a certain condition. In the shell, the statement uses the following forms, much like the one used in other languages.

| if command is successful then execute commands else execute commands fi | if command is successful then execute commands fi | if command is successful then execute commands elif command is successful then... else... fi |
|---|---|---|
| **Form 1** | **Form 2** | **Form 3** |

As in BASIC, **if** also requires a **then**. It evaluates the success or failure of the *command* that is specified in its "command line." If *command* succeeds, the sequence of commands following it is executed. If *command* fails, then the **else** statement (if present) is executed. This statement is not always required, as shown in Form 2. Every **if** is closed with a corresponding **fi**, and you'll encounter an error if one is not present.

What makes shell programming so powerful is that a command's exit status solely determines the course of action pursued by many of the shell's important constructs like **if** and **while**. All commands return an exit status as we saw with **cat** and **grep**, so you can imagine where shell programming can lead us.

In the next script, **emp3.sh** (Fig. 14.4), **grep** is first executed and a simple **if-else** construct tests the exit status of **grep**. This time we'll search /etc/passwd for the existence of two users; one exists in the file and the other doesn't:

```
$ emp3.sh ftp
ftp:*:325:15:FTP User:/users1/home/ftp:/bin/true
Pattern found - Job Over
$ emp3.sh mail
Pattern not found
```

```
#!/bin/sh
# emp3.sh: Using if and else
#
if grep "^$1" /etc/passwd 2>/dev/null    # Search username at beginning of line
then
    echo "Pattern found - Job Over"
else
    echo "Pattern not found"
fi
```

Fig. 14.4  emp3.sh

We'll discuss the third form of the **if** statement when we discuss **test**. The condition placed in the command line of the **if** statement will henceforth be referred to as the **control command**. You can use **if** in this way with any executable program. Amazing power indeed!

## 14.7 USING test AND [] TO EVALUATE EXPRESSIONS

When you use **if** to evaluate expressions, you need the **test** statement because the true or false values returned by expressions can't be *directly* handled by **if**. **test** uses certain operators to evaluate the condition on its right and returns either a true or false exit status, which is then used by **if** for making decisions. **test** works in three ways:

- Compares two numbers.
- Compares two strings or a single one for a null value.
- Checks a file's attributes.

These tests can be made by **test** in association with the shell's other statements also, but for the present we'll stick with **if**. **test** doesn't display any output but simply sets the parameter $?. In the following sections, we'll check this value.

### 14.7.1 Numeric Comparison

The numerical comparison operators (Table 14.2) used by **test** have a form different from what you would have seen anywhere. They always begin with a - (hyphen), followed by a two-letter string, and enclosed on either side by whitespace. Here's a typical operator:

-ne     Not equal

The operators are quite mnemonic; -eq implies equal to, -gt implies greater than, and so on. Numeric comparison in the shell is confined to integer values only; decimal values are simply truncated. To illustrate how numeric tests are performed, we'll assign some values to three variables and numerically compare them:

```
$ x=5; y=7; z=7.2
$ test $x -eq $y ; echo $?
1
$ test $x -lt $y ; echo $?                    Not equal
0
$ test $z -gt $y ; echo $?                    True
1
$ test $z -eq $y ; echo $?                    7.2 is not greater than 7!
0
                                              7.2 is equal to 7!
```

The last two tests prove conclusively that numeric comparison is restricted to integers only. Having used **test** as a standalone feature, you can now use it as **if**'s control command. The next script, emp3a.sh (Fig. 14.5) uses **test** in an **if-elif-else-fi** construct (Form 3) to evaluate the shell parameter, $#. It displays the usage when no arguments are input, runs **grep** if two arguments are entered and displays an error message otherwise.

```
#!/bin/sh
# emp3a.sh: Using test, $0 and $# in an if-elif-if construct
#
if test $# -eq 0 ; then
     echo "Usage: $0 pattern file" >/dev/tty
elif test $# -eq 2 ; then
     grep "$1" $2 || echo "$1 not found in $2" >/dev/tty
else
     echo "You didn't enter two arguments" >/dev/tty
fi
```

**Fig. 14.5 emp3a.sh**

Why did we redirect the **echo** output to /dev/tty? Simple, we want the script to work both with and without redirection. In either case, the output of the **echo** statements must appear *only* on the terminal. These statements are used here as "error" messages even though they are not directed to the standard error. Now run the script four times and redirect the output every time:

```
$ emp3a.sh > foo
Usage: emp3a.sh pattern file
$ emp3a.sh ftp > foo
You didn't enter two arguments
$ emp3a.sh henry /etc/passwd > foo
henry not found in /etc/passwd
$ emp3a.sh ftp /etc/passwd > foo
$ cat foo
ftp:*:325:15:FTP User:/users1/home/ftp:/bin/true
```

The importance of /dev/tty as a mechanism of explicitly redirecting an output stream shows up in this example. You must appreciate this and use this feature in shell scripts when you like to have the redirection option open. The above script works just as well even if you don't redirect it.

*Tip: An application may need to be designed in a flexible manner to allow redirection of an entire script or its participation in a pipeline. In that case, you need to ensure that messages meant to draw the attention of the user (mainly from **echo**) are redirected to >/dev/tty.*

**Table 14.2 Numerical Comparison Operators Used by test**

| Operator | Meaning |
|---|---|
| -eq | Equal to |
| -ne | Not equal to |
| -gt | Greater than |
| -ge | Greater than or equal to |
| -lt | Less than |
| -le | Less than or equal to |

The **test** output is true only if both variables are nonnull strings, i.e., the user enters some nonwhitespace characters when the script pauses twice.

---

**Tip:** Observe that we have been quoting our variables wherever possible. Quoting is essential when you assign multiple words to a variable. To try that out, drop the quotes to use the statement `if [ -z $pname ]`. When you input two words, or even a null string to be assigned to pname, you'll often encounter an error. Quoting is safe with no adverse consequences.

---

**Table 14.3** String Tests Used by **test**

| Test | True if |
|------|---------|
| s1 = s2 | String s1 = s2 |
| s1 != s2 | String s1 is not equal to s2 |
| -n stg | String stg is not a null string |
| -z stg | String stg is a null string |
| stg | String stg is assigned and not null |
| s1 == s2 | String s1 = s2 (Korn and Bash only) |

## 14.7.3 File Tests

**test** can be used to test the various file attributes like its type (file, directory or symbolic link) or its permissions (read, write, execute, SUID, etc.). Both **perl** and the UNIX system call library also offer these facilities (Table 14.4). Let's test some attributes of the file emp.1st at the prompt:

```
$ ls -l emp.1st
-rw-rw-rw-   1 kumar      group        870 Jun  8 15:52 emp.1st
$ [ -f emp.1st ] ; echo $?                          An ordinary file?
0                                                   Yes
$ [ -x emp.1st ] ; echo $?                          An executable file?
1                                                   No
$ [ ! -w emp.1st ] || echo "False that file is not writable"
False that file is not writable
```

The **!** negates a test, so `[ ! -w foo ]` negates `[ -w foo ]`. Using these features, you can design a script, **filetest.sh** (Fig. 14.7), that accepts a filename as argument and then performs a number of tests on it. Test the script with two filenames—one that doesn't exist and one that does:

```
$ filetest.sh emp3.1st
File does not exist
$ filetest.sh emp.1st
File is both readable and writable
```

This completes the discussion on the three domains of **test**—numeric comparison, testing of strings and file attributes. Even though we used **test** with the **if** statement in all of our examples, **test** returns an **exit** status only, and can thus be used with any shell construct that uses an exit status. **test** also finds wide application in the **while** statement.

```
#!/bin/sh
# filetest.sh: Tests file attributes
#
if [ ! -e $1 ] ; then
    echo "File does not exist"
elif [ ! -r $1 ] ; then
    echo "File is not readable"
elif [ ! -w $1 ] ; then
    echo "File is not writable"
else
    echo "File is both readable and writable"
fi
```

**Fig. 14.7 filetest.sh**

Table 14.4  File-related Tests with **test**

| Test | True if File |
|---|---|
| -f *file* | *file* exists and is a regular file |
| -r *file* | *file* exists and is readable |
| -w *file* | *file* exists and is writable |
| -x *file* | *file* exists and is executable |
| -d *file* | *file* exists and is a directory |
| -s *file* | *file* exists and has a size greater than zero |
| -e *file* | *file* exists (Korn and Bash only) |
| -u *file* | *file* exists and has SUID bit set |
| -k *file* | *file* exists and has sticky bit set |
| -L *file* | *file* exists and is a symbolic link (Korn and Bash only) |
| *f1* -nt *f2* | *f1* is newer than *f2* (Korn and Bash only) |
| *f1* -ot *f2* | *f1* is older than *f2* (Korn and Bash only) |
| *f1* -ef *f2* | *f1* is linked to *f2* (Korn and Bash only) |

## 14.8 THE case CONDITIONAL

The **case** statement is the second conditional offered by the shell. It doesn't have a parallel either in C (**switch** is similar) or **perl**. The statement matches an expression for more than one alternative, and uses a compact construct to permit multiway branching. **case** also handles string tests, but in a more efficient manner than **if**. The general syntax of the **case** statement is as follows:

```
case expression in
    pattern1) commands1 ;;
    pattern2) commands2 ;;
    pattern3) commands3 ;;
    .....
esac
```

```
#!/bin/sh
# menu.sh: Uses case to offer 5-item menu
#
echo "              MENU\n
1. List of files\n2. Processes of user\n3. Today's Date
4. Users of system\n5. Quit to UNIX\nEnter your option: \c"
read choice
case "$choice" in
    1) ls -l ;;
    2) ps -f ;;
    3) date  ;;
    4) who   ;;
    5) exit  ;;
    *) echo "Invalid option"          # ;; not really required for the last option
esac
```

**Fig. 14.8  menu.sh**

**case** first matches *expression* with *pattern1*. If the match succeeds, then it executes *command* which may be one or more commands. If the match fails, then *pattern2* is matched, and so on. Each command list is terminated with a pair of semicolons, and the entire construct is closed with **esac** (reverse of **case**).

Consider a simple script, **menu.sh** (Fig. 14.8) that uses **case**. The script accepts values from 1 to 5 and performs some action depending on the number keyed in. The five menu choices are displayed with a multi-line **echo** statement.

**case** matches the value of $choice with the strings 1, 2, 3, 4 and 5. If the user enters a 1, the ls -l command is executed. Option 5 quits the program. The last option (*) matches any option not matched by by the previous options. We'll make good use of the * later.

You can see today's date by choosing the third option:

```
$ menu.sh
          MENU

1. List of files
2. Processes of user
3. Today's Date
4. Users of system
5. Quit to UNIX
Enter your option: 3
Tue Jan  7 18:03:06 IST 2003
```

**case** can't handle relational and file tests, but it matches strings with compact code. It is also effective when the string is fetched by command substitution. If you cut our the first field from the date output, you can use this **case** construct to do different things, depending on the day of the week:

```
case `date | cut -d" " -f1` in
```

*Outputs three-character day string*

```
Mon) tar -cvf /dev/fd0 $HOME ;;
Wed) scp -r $HOME mercury:/home/henry ;;
Fri) find $HOME -newer .last_full_backup_time -print > tarilist ;;
  *) ;;
```
esac

**case** can also handle numbers, but only by treating them as strings. Some of our previous programs used **if** and **test** to check the value of $#. You can use **case** to match $# *directly* (without using **test**) with specific values (0, 1, 2, etc.), but you can't use it to make numeric checks (of the type $# -gt 2).

## 14.8.1 Matching Multiple Patterns

**case** can also specify the same action for more than one pattern. Programmers frequently encounter a logic that has to test a user response for both y and Y (or n and N). Like **grep -E** and **egrep**, **case** uses the | to delimit multiple patterns. For instance, the expression y|Y can be used to match y in both upper and lowercase:

```
echo "Do you wish to continue? (y/n): \c"
read answer
case"$answer" in
    y|Y) ;;                 Null statement, no action to be performed
    n|N) exit ;;
esac
```

The same logic would require a larger number of lines if implemented with **if**. **case** becomes an automatic choice when the number of matching options is high.

## 14.8.2 Wild-Cards: case Uses Them

**case** has a superb string matching feature that uses wild-cards. It uses the filename matching metacharacters *, ? and the character class (8.3.3)—*but only to match strings and not files in the current directory*. The revised **case** construct of a previous example lets the user answer the question in several ways:

```
case "$answer" in
    [yY][eE]*) ;;           Matches YES, yes, Yes, YEs, yES, etc.
    [nN][oO]) exit ;;       Matches NO, no, nO and No
           *) echo "Invalid response"When everything else fails
esac
```

Wild-card usage in the first two options appears simple enough. Note that the * appears in two options. In the first option, it behaves like a normal wild-card. In the last option, it provides a refuge for all other nonmatched options. Note that the last **case** option doesn't need ;; but you can provide them if you want.

## 14.9 expr: COMPUTATION AND STRING HANDLING

The Bourne shell can check whether an integer is greater than another, but it doesn't have any computing features at all. It has to rely on the external **expr** command for that purpose. This command combines two functions in one:

- Performs arithmetic operations on integers.
- Manipulates strings.

We'll use **expr** to perform both these functions, but with not-very-readable code when it comes string handling. If you are using the Korn shell or Bash, you have better ways of handling the things (21.7), but you must also understand the helplessness of Bourne. It's quite possible that you have to debug someone else's script which contains **expr**.

## 14.9.1 Computation

**expr** can perform the four basic arithmetic operations as well as the modulus (remainder) function

```
$ x=3 y=5                          Multiple assignments without a ;
$ expr 3 + 5
8
$ expr $x - $y
-2
$ expr 3 \* 5                      Asterisk has to be escaped
15
$ expr $y / $x
1
                                   Decimal portion truncated
$ expr 13 % 5
3
```

The operand, be it +, -, * etc., must be enclosed on either side by whitespace. Observe that the multiplication operand (*) has to be escaped to prevent the shell from interpreting it as the filename metacharacter. Since **expr** can handle only integers, division yields only the integral part.

**expr** is often used with command substitution to assign a variable. For example, you can set a variable z to the sum of two numbers:

```
$ x=6 y=2 ; z=`expr $x + $y`
$ echo $z
8
```

Perhaps the most common use of **expr** is in incrementing the value of a variable. All programming languages have a shorthand method of doing that, and it is natural that UNIX should also have its own

```
$ x=5
$ x=`expr $x + 1`                  This is the same as C's x++
$ echo $x
6
```

If you are using the Korn shell or Bash, then you can turn to Section 21.5 for a discussion on the **let** statement that both shells use to handle computation.

## 14.9.2 String Handling

Though **expr**'s string handling facilities aren't exactly elegant, Bourne shell users hardly have a choice. For manipulating strings, **expr** uses two expressions separated by a colon. The string to be worked upon is placed on the left of the :, and a regular expression is placed on its right. Depending on the composition of the expression, **expr** can perform three important string functions.

- Determine the length of the string.
- Extract a substring.
- Locate the position of a character in a string.

**The Length of a String**    The length of a string is a relatively simple matter, the regular expression * signifies to **expr** that it has to print the number of characters matching the pattern, i.e., the length of the entire string:

```
$ expr "abcdefghijkl" : '.*'                    Space on either side of : required
12
```

Here, **expr** has counted the number of occurrences of any character (.*). This feature is useful in validating data entry. Consider that you want to validate the name of a person accepted through the keyboard so that it doesn't exceed, say, 20 characters in length. The following **expr** sequence can be quite useful for this task:

```
while echo "Enter your name: \c" ; do            echo always returns true
    read name
    if [ `expr "$name" : '.*'` -gt 20 ] ; then
        echo "Name too long"
    else
        break                                    break terminates a loop
    fi
done
```

**Extracting a Substring**    **expr** can extract a string enclosed by the escaped characters \( and \). If you wish to extract the 2-digit year from a 4-digit string, you must create a pattern group and extract it this way:

```
$ stg=2003
$ expr "$stg" : '..\(..\)'                       Extracts last two characters
03
```

Note the pattern group \(..\). This is the tagged regular expression (TRE) used by **sed** (13.11.3), but it is used here with a somewhat different meaning. It signifies that the first two characters in the value of $stg have to be ignored and two characters have to be extracted from the third character position. (There's no \1 and \2 here.)

**Locating Position of a Character**    **expr** can also return the location of the first occurrence of a character inside a string. To locate the position of the character d in the string value of $stg, you have to count the number of characters which are not d ([^d]*), followed by a d:

```
$ stg=abcdefgh ; expr "$stg" : '[^d]*d'
4
```

**expr** duplicates some of the features of the **test** statement, and also uses the relational operators in the same way. They are not pursued here because **test** is a built-in feature of the shell, and is consequently faster. The Korn shell and Bash have built-in string handling facilities; they don't need **expr**. These features are taken up in Chapter 21.

# 14.11 while: LOOPING

None of the pattern scanning scripts developed so far offers the user another chance to rectify a faulty response. Loops let you perform a set of instructions repeatedly. The shell features three types of loops—**while**, **until** and **for**. All of them repeat the instruction set enclosed by certain keywords as often as their control command permits.

The **while** statement should be quite familiar to most programmers. It repeatedly performs a set of instructions until the control command returns a true exit status. The general syntax of this command is as follows:

while *condition is true*

do

   *commands*                         *Note the* do *keyword*

done                         *Note the* done *keyword*

The *commands* enclosed by **do** and **done** are executed repeatedly as long as *condition* remains true. You can use any UNIX command or **test** as the *condition*, as before.

We'll start with an orthodox **while** loop application. The script, **emp5.sh** Fig. 14.10), accepts a code and description in the same line, and then writes the line to **newlist**. It then prompts you for more entries. The loop iteration is controlled by the value of $answer.

We have redirected the output of two **echo** statements to /dev/tty for reasons that will be apparent later. We'll make a small but significant modification later, but let's run it first:

```
$ emp5.sh
Enter the code and description: 03 analgesics
Enter any more (y/n)? y
```

```
#!/bin/sh
# emp5.sh: Shows use of the while loop
#
answer=y          # Must set it to y first to enter the loop
while [ "$answer" = "y" ]      # The control command
do
    echo "Enter the code and description: \c" >/dev/tty
    read code description      # Read both together
    echo "$code|$description" >> newlist      # Append a line to newlist
    echo "Enter any more (y/n)? \c" >/dev/tty
    read anymore
    case $anymore in
        y*|Y*) answer=y ;;      # Also accepts yes, YES etc.
        n*|N*) answer=n ;;      # Also accepts no, NO etc.
            *) answer=y ;;      # Any other reply means y
    esac
done
```

Fig. 14.10 **emp5.sh**